Metody numeryczne II.
Błędy reprezentacji liczb

Oleksandr Sokolov

Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK
http://fizyka.umk.pl/~osokolov/MNII/

Reprezentacja liczb

- Reprezentacja **stałopozycyjna**

13, -145.23

Reprezentacja stałopozycyjna operuje na **ustalonej liczbie cyfr ułamkowych** – wszystkie liczby rzeczywiste skraca się do t cyfr ułamkowych.

- Reprezentacja **zmiennopozycyjna**

mantysa i cecha,

liczba = mantysa $*10^{cecha}$

$18.5=0.185*10^2=1.85*10^1=185*10^{-1}$

Cecha wskazuje miejsce kropki w zapisie liczbowym

Reprezentację zmiennopozycyjną liczby x oznaczamy

rd(x) –rounding decimals

Reprezentacja stałoprzecinkowa

- <u>Wady reprezentacji stałoprzecinkowej (***Fixed Point Notation***):</u>

$$B = 0.000075244$$

$$A = 47567.31$$

- 10-cyfrowy format: XXXXX.XXXXX

$$A = 47567.31\boxed{000}$$

$$B = 0.00007\boxed{5244} =$$
$$= 00000.00007$$

W przypadku liczb stałoprzecinkowych wystąpi **duży błąd przy bardzo małych wartościach** oraz **bardzo dużych wartościach** (w odniesieniu do powyższego formatu).

Reprezentacja stałoprzecinkowa (cd.)

Kody stałopozycyjne mają ***ustalone miejsce rozdziału części całkowitej i ułamkowej***, czyli miejsce przecinka, co oznacza, że **dokładność reprezentacji jest stała**.

Aby mówić o arytmetyce stałoprzecinkowej trzeba wiedzieć, w jaki sposób zapisywane są **liczby dodatnie i ujemne**. Dla reprezentacji liczb dodatnich i ujemnych najpowszechniej stosowane są **trzy kody**:

- kod znak – moduł (ZM)
- kod uzupełnienia do jednego (U1)
- kod uzupełnienia do dwóch (U2)

Reprezentacja stałoprzecinkowa (cd.)

4 bitowy system ZM		
Kod ZM	Przeliczenie	Wartość
0000	$(-1)^0 \times 0$	0
0001	$(-1)^0 \times (2^0)$	1
0010	$(-1)^0 \times (2^1)$	2
0011	$(-1)^0 \times (2^1 + 2^0)$	3
0100	$(-1)^0 \times (2^2)$	4
0101	$(-1)^0 \times (2^2 + 2^0)$	5
0110	$(-1)^0 \times (2^2 + 2^1)$	6
0111	$(-1)^0 \times (2^2 + 2^1 + 2^0)$	7
1000	$(-1)^1 \times 0$	0
1001	$(-1)^1 \times (2^0)$	(-1)
1010	$(-1)^1 \times (2^1)$	(-2)
1011	$(-1)^1 \times (2^1 + 2^0)$	(-3)
1100	$(-1)^1 \times (2^2)$	(-4)
1101	$(-1)^1 \times (2^2 + 2^0)$	(-5)
1110	$(-1)^1 \times (2^2 + 2^1)$	(-6)
1111	$(-1)^1 \times (2^2 + 2^1 + 2^0)$	(-7)

4 bitowy system U1		
Kod U1	Przeliczenie	Wartość
0000	0	0
0001	2^0	1
0010	2^1	2
0011	$2^1 + 2^0$	3
0100	2^2	4
0101	$2^2 + 2^0$	5
0110	$2^2 + 2^1$	6
0111	$2^2 + 2^1 + 2^0$	7
1000	$(-2^3 + 1)$	(-7)
1001	$(-2^3 + 1) + 2^0$	(-6)
1010	$(-2^3 + 1) + 2^1$	(-5)
1011	$(-2^3 + 1) + 2^1 + 2^0$	(-4)
1100	$(-2^3 + 1) + 2^2$	(-3)
1101	$(-2^3 + 1) + 2^2 + 2^0$	(-2)
1110	$(-2^3 + 1) + 2^2 + 2^1$	(-1)
1111	$(-2^3 + 1) + 2^2 + 2^1 + 2^0$	0

4 bitowy system U2		
Kod U2	Przeliczenie	Wartość
0000	0	0
0001	2^0	1
0010	2^1	2
0011	$2^1 + 2^0$	3
0100	2^2	4
0101	$2^2 + 2^0$	5
0110	$2^2 + 2^1$	6
0111	$2^2 + 2^1 + 2^0$	7
1000	(-2^3)	(-8)
1001	$(-2^3) + 2^0$	(-7)
1010	$(-2^3) + 2^1$	(-6)
1011	$(-2^3) + 2^1 + 2^0$	(-5)
1100	$(-2^3) + 2^2$	(-4)
1101	$(-2^3) + 2^2 + 2^0$	(-3)
1110	$(-2^3) + 2^2 + 2^1$	(-2)
1111	$(-2^3) + 2^2 + 2^1 + 2^0$	(-1)

System ZM

4 bitowy system ZM		
Kod ZM	**Przeliczenie**	**Wartość**
0000	$(-1)^0 \times 0$	0
0001	$(-1)^0 \times (2^0)$	1
0010	$(-1)^0 \times (2^1)$	2
0011	$(-1)^0 \times (2^1 + 2^0)$	3
0100	$(-1)^0 \times (2^2)$	4
0101	$(-1)^0 \times (2^2 + 2^0)$	5
0110	$(-1)^0 \times (2^2 + 2^1)$	6
0111	$(-1)^0 \times (2^2 + 2^1 + 2^0)$	7
1000	$(-1)^1 \times 0$	0
1001	$(-1)^1 \times (2^0)$	(-1)
1010	$(-1)^1 \times (2^1)$	(-2)
1011	$(-1)^1 \times (2^1 + 2^0)$	(-3)
1100	$(-1)^1 \times (2^2)$	(-4)
1101	$(-1)^1 \times (2^2 + 2^0)$	(-5)
1110	$(-1)^1 \times (2^2 + 2^1)$	(-6)
1111	$(-1)^1 \times (2^2 + 2^1 + 2^0)$	(-7)

$$10110111_{(ZM)} = (-1)^1 \times (2^5 + 2^4 + 2^2 + 2^1 + 2^0)$$
$$10110111_{(ZM)} = -(32 + 16 + 4 + 2 + 1)$$
$$10110111_{(ZM)} = -55_{(10)}$$

$$00011111_{(ZM)} = (-1)^0 \times (2^4 + 2^3 + 2^2 + 2^1 + 2^0)$$
$$00011111_{(ZM)} = (16 + 8 + 4 + 2 + 1)$$
$$00011111_{(ZM)} = 31_{(10)}$$

System U1

$$b_{n-1}b_{n-2}b_{n-3}\ldots b_2b_1b_{0\,(U1)} = b_{n-1}(-2^{n-1}+1) + b_{n-2}2^{n-2} + b_{n-3}2^{n-3} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

4 bitowy system U1		
Kod U1	**Przeliczenie**	**Wartość**
0000	0	0
0001	2^0	1
0010	2^1	2
0011	$2^1 + 2^0$	3
0100	2^2	4
0101	$2^2 + 2^0$	5
0110	$2^2 + 2^1$	6
0111	$2^2 + 2^1 + 2^0$	7
1000	$(-2^3 + 1)$	(-7)
1001	$(-2^3 + 1) + 2^0$	(-6)
1010	$(-2^3 + 1) + 2^1$	(-5)
1011	$(-2^3 + 1) + 2^1 + 2^0$	(-4)
1100	$(-2^3 + 1) + 2^2$	(-3)
1101	$(-2^3 + 1) + 2^2 + 2^0$	(-2)
1110	$(-2^3 + 1) + 2^2 + 2^1$	(-1)
1111	$(-2^3 + 1) + 2^2 + 2^1 + 2^0$	0

Liczba przeciwna zawsze powstaje w kodzie U1 przez **negację** wszystkich bitów

$1_{(10)} = 0001_{(U1)} : (-1)_{(10)} = 1110_{(U1)}$

$5_{(10)} = 0101_{(U1)} : (-5)_{(10)} = 1010_{(U1)}$

$7_{(10)} = 0111_{(U1)} : (-7)_{(10)} = 1000_{(U1)}$

Jeśli liczba jest dodatnia, to najstarszy bit znakowy posiada wartość **0**. Pozostałe bity służą do zapisu liczby w naturalnym kodzie binarnym:

$0111_{(U1)} = 7_{(10)}$, $0001_{(U1)} = 1_{(10)}$, $01111111_{(U1)} = 127_{(10)}$

Jeśli liczba jest ujemna, to najstarszy bit znakowy ma wartość **1**. Pozostałe bity są **negacjami** bitów modułu wartości liczby:

$1101_{(U1)} = (-2)_{(10)}$: moduł $2_{(10)} = 010_{(2)}$: **NOT** 010 = 101

$1100_{(U1)} = (-3)_{(10)}$: moduł $3_{(10)} = 011_{(2)}$: **NOT** 011 = 100

$1010_{(U1)} = (-5)_{(10)}$: moduł $5_{(10)} = 101_{(2)}$: **NOT** 101 = 010

System U2

W tym systemie wartość **0 ma tylko jedną reprezentację** 0000, a liczb ujemnych jest o 1 więcej niż dodatnich (-8 .. -1, 1 .. 7).

4 bitowy system U2		
Kod U2	Przeliczenie	Wartość
0000	0	0
0001	2^0	1
0010	2^1	2
0011	$2^1 + 2^0$	3
0100	2^2	4
0101	$2^2 + 2^0$	5
0110	$2^2 + 2^1$	6
0111	$2^2 + 2^1 + 2^0$	7
1000	(-2^3)	(-8)
1001	$(-2^3) + 2^0$	(-7)
1010	$(-2^3) + 2^1$	(-6)
1011	$(-2^3) + 2^1 + 2^0$	(-5)
1100	$(-2^3) + 2^2$	(-4)
1101	$(-2^3) + 2^2 + 2^0$	(-3)
1110	$(-2^3) + 2^2 + 2^1$	(-2)
1111	$(-2^3) + 2^2 + 2^1 + 2^0$	(-1)

Najstarszy bit określa znak liczby. Jeśli jest równy 0, liczba jest dodatnia i resztę zapisu możemy potraktować jak liczbę w naturalnym kodzie dwójkowym.

$$01101011_{(U2)} = 64 + 32 + 8 + 2 + 1 = 107_{(10)}.$$

Jeśli bit znaku ustawiony jest na 1, to liczba ma wartość ujemną. Bit znaku ma wagę (-2^{n-1}), gdzie **n** oznacza liczbę bitów w wybranym formacie U2. Reszta bitów jest zwykłą liczbą w naturalnym kodzie dwójkowym. Wagę bitu znakowego i wartość pozostałych bitów sumujemy otrzymując wartość liczby U2:

$$11101011_{(U2)} = (-2^7) + 64 + 32 + 8 + 2 + 1 = -128 + 107 = (-21)_{(10)}$$

Postać ujemna liczby U2 nie jest już tak czytelna dla nas jak w przypadku kodów ZM (tylko zmiana bitu znaku) i U1 (negacja wszystkich bitów).

Typy danych w języku C++

eduinf.waw.pl/inf/alg/006_bin/0027.php

Liczby ze znakiem

short, short int, signed short

2-bajtowa (16 bitów) liczba całkowita ze znakiem w kodzie uzupełnieniowym do 2 - U2.
Zakres od -2^{15} (-32768) do $2^{15} - 1$ (32767).

int, signed int, signed

4-bajtowa liczba całkowita ze znakiem w kodzie **U2** (32 bity).
Zakres od -2^{31} (-2147483648) do $2^{31} - 1$ (2147483647).

long, signed long int, signed long

4-bajtowa liczba całkowita ze znakiem w kodzie **U2** (32 bity).
Zakres od -2^{31} (-2147483648) do $2^{31} - 1$ (2147483647).

long long, long long int, signed long long, signed long long int

8-bajtowa liczba całkowita ze znakiem w kodzie **U2** (64 bity).
Zakres od -2^{63} (-9223372036854775808) do $2^{63} - 1$ (9223372036854775807).

Podsumowanie

Typy całkowite

https://eduinf.waw.pl/inf/alg/006_bin/0027.php

Zapis zmiennopozycyjny

Z zapisem zmiennoprzecinkowym można spotkać się w przypadkach, gdzie przy jego pomocy przedstawia się albo bardzo duże wartości, albo bardzo małe. Zapis ten nazywa się często **notacją naukową**, np.:

Gwiazda Proxima Centauri znajduje się w odległości **9460800000000** [km], czyli **9,4608** x $\mathbf{10}^{12}$.

Masa elektronu wynosi m_e = **0,000000000000000000000000091095** [g], czyli **9,1095** x $\mathbf{10}^{-28}$ [g]

Podstawa matematyczna

Liczbę rzeczywistą w komputerze reprezentuje liczba zmiennoprzecinkowa:

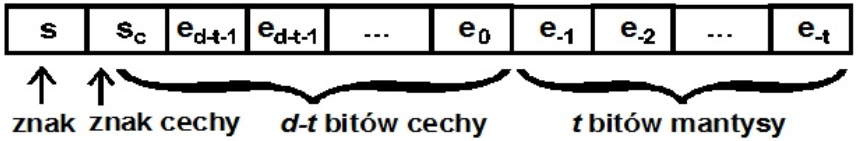
$$F = M\beta^E$$

gdzie:
M – znacznik („mantysa”) jest liczbą ułamkową ze znakiem
β – stanowi bazę reprezentacji i jest liczbą całkowitą (np.: 2, 10, 16)
E – wykładnik („cecha”) jest znakowaną liczbą całkowitą

Reprezentacja binarna

Zaokrąglanie ułamków dziesiętnych

rounding **d**ecimals



$$rd(x) = sm_t \cdot 2^c$$

$$m_t = \sum_{i=1}^{t} e_{-i} \cdot 2^{-i}$$

Przykład

d=6,t=3

0 0 0 0 1 0 1

$Rd(x)=(e_{-1}*2^{-1}+e_{-2}*2^{-2}+e_{-3}*2^{-3})*2^0$

$Rd(x)=(1*2^{-1}+0*2^{-2}+1*2^{-3})*2^0$

Rd(x)=5/8=0.625

Błędy reprezentacji

$$m = \sum_{i=1}^{\infty} e_{-i} \cdot 2^{-i} \quad \left(e_{-1} = 1; e_{-i} = 0 \text{ lub } 1 \text{ dla } i > 1\right)$$

$$rd(x) = sm_t \cdot 2^c$$

Błąd bezwzględny reprezentacji

$$\varepsilon = rd(x) - x$$

Błąd względny reprezentacji binarnej

$$r = \left| \frac{rd(x) - x}{x} \right| \le 2^{-t}$$

$$rd(x) = x \cdot (1 + r)$$

$$\tilde{x} = x \pm \varepsilon$$

$$r = \frac{\varepsilon}{x}$$

$$\varepsilon = rx$$

$$\tilde{x} = x + \varepsilon = x + rx = x(1 + r)$$

Przykład

d=6,t=3

0 0 0 0 1 **0 1**

Rd(x)=$(1*2^{-1}+0*2^{-2}+1*2^{-3})*2^0$

Rd(x)=5/8=0.625

Rd(y)=$(1*2^{-1}+0*2^{-2}+0*2^{-3})*2^0$

Rd(y)=1/2=0.5

Rd(z)=$(1*2^{-1}+1*2^{-2}+0*2^{-3})*2^0$

Rd(z)=3/4=0.75

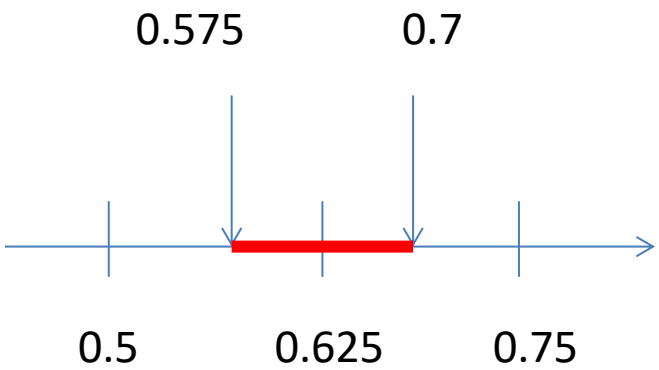
Przykład

$$x = 0.6$$
$$\boxed{\tilde{x} = 0.625}$$
$$\varepsilon = 0.025$$
$$r = \frac{0.025}{0.6} = 0.0417 \le 2^{-3} = 0.125$$

$$x = 0.6$$
$$\tilde{x} = 0.5$$
$$\varepsilon = 0.1$$
$$r = \frac{0.1}{0.5} = 0.2 > 2^{-3} = 0.125$$

0.025<1/2*10^-**1** – jedna cyfra poprawna

$$x = 0.625$$
$$r = 2^{-3} = 0.125$$
$$\varepsilon = rx = 0.075$$
$$\tilde{x} = 0.625 \pm 0.075$$



Cechy reprezentacji

Liczba cyfr mantysy decyduje o dokładności zmiennopozycyjnego przedstawiania liczb, a **liczba cyfr cechy** określa zakres reprezentowalnych liczb

$$\frac{1}{2} \cdot 2^{c_{\min}} \leq |x| < 2^{c_{\max}}$$

$$c_{\min} = -2^{d-t} + 1$$

$$c_{\max} = 2^{d-t} - 1$$

Przykład

d=6,t=3

0 0 0 0 1 0 1

$Rd(x)=(e_{-1}*2^{-1}+e_{-2}*2^{-2}+e_{-3}*2^{-3})*2^{0}$

$Rd(x)=(1*2^{-1}+0*2^{-2}+1*2^{-3})*2^{0}$

Rd(x)=5/8=0.625

Postać znormalizowana liczby

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-1}}{\beta^{t-1}}\right)\beta^e$$

$$0 \le d_i \le \beta - 1,\ i = 1, \ldots, t-1$$

$$0 < d_0 \le \beta - 1$$

!

$$L \le e \le U$$

$$\boxed{\beta = 10}$$

$$\boxed{\beta = 2}$$

$$\boxed{\beta = 8}$$

$$\boxed{\beta = 16}$$

Standardy

System	β	t	L	U
IEEE SP	2	24	-126	127
IEEE DP	2	53	$-1,022$	$1,023$
Cray	2	48	$-16,383$	$16,384$
HP calculator	10	12	-499	499
IBM mainframe	16	6	-64	63

$$a = 0.1234567 \cdot 10^0, b = 0.4711325 \cdot 10^4$$

Przykład

$$\boxed{\beta = 2, L = -1, U = 1, t = 3}$$

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-1}}{\beta^{t-1}} \right)\beta^e$$

$$0 \le d_i \le \beta - 1,\ i = 1, \ldots, t-1$$

$$0 < d_0 \le \beta - 1$$

$$L \le e \le U$$

Przykład (cd)

d={0,1}
d0=1
X=(d0+d1/2+d2/4)*2^{-1,0,1}

1 0 0 2^-1
1 0 0 2^0
1 0 0 2^1

1 0 1 2^-1
1 0 1 2^0
1 0 1 2^1

1 1 0 2^-1
1 1 0 2^0
1 1 0 2^1

1 1 1 2^-1
1 1 1 2^0
1 1 1 2^1

3*8=12 wartości
liczb dodatnich

$$\beta = 2, L = -1, U = 1, t = 3$$

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-1}}{\beta^{t-1}}\right)\beta^e$$

$$0 \le d_i \le \beta - 1,\ i = 1, \ldots, t-1$$

$$0 < d_0 \le \beta - 1$$

$$L \le e \le U$$

Przykład(cd)

d={0,1}
d0=1
X=(d0+d1/2+d2/4)*2^{-1,0,1}

1 0 0 2^-1
1 0 0 2^0
1 0 0 2^1

1 0 1 2^-1
1 0 1 2^0
1 0 1 2^1

1 1 0 2^-1
1 1 0 2^0
1 1 0 2^1

1 1 1 2^-1
1 1 1 2^0
1 1 1 2^1

3*8=12 wartości liczb dodatnich

krok

0.125

0.25

0.5

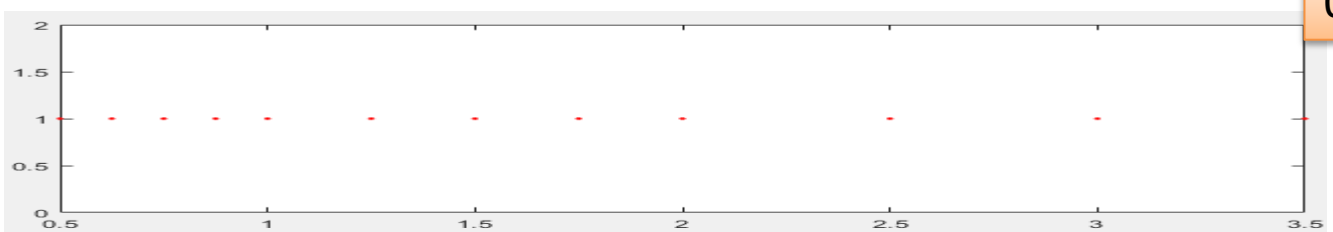
0.5
0.625
0.75
0.875
1
1.25
1.5
1.75
2
2.5
3
3.5

1 0 0 2^-1
1 0 0 2^0
1 0 0 2^1

X=1*2^-1=0.5
X=1*2^0=1
X=1*2^1=2

1 0 1 2^-1
1 0 1 2^0
1 0 1 2^1

X=(1+1/4)*2^-1=0.625
X=(1+1/4)*2^0=1.25
X=(1+1/4)*2^1=2.5



Cechy liczb zmiennopozycyjnych

Ilość możliwych wartości

$$2(\beta-1)\beta^{t-1}(U-L+1)$$

2*1*2^2*(1+1+1)=24

Najmniejsza wartość
Underflow level

$$UFL=\beta^{L}$$

2^-1=0.5

Największa wartość
Overflow level

$$OFL=\beta^{U+1}\left(1-\beta^{-t}\right)$$

2^2*(1-1/8)=3.5

0.5
0.625
0.75
0.875
1
1.25
1.5
1.75
2
2.5
3
3.5

$$\beta=2, L=-1, U=1, t=3$$

Zaokrąglenia

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-2}}{\beta^{t-2}} + \frac{d_{t-1}}{\beta^{t-1}} \right)\beta^e$$

obcięcie

$$fl(x) = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-2}}{\beta^{t-2}} \right)\beta^e$$

Zaokrąglenie do najbliższej

$$fl(x) = \begin{cases} \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-2}}{\beta^{t-2}} \right)\beta^e \\ \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-2}+1}{\beta^{t-2}} \right)\beta^e \end{cases}$$

Number	Chop	Round to nearest	Number	Chop	Round to nearest
1.649	1.6	1.6	1.749	1.7	1.7
1.650	1.6	1.6	1.750	1.7	1.8
1.651	1.6	1.7	1.751	1.7	1.8
1.699	1.6	1.7	1.799	1.7	1.8

Epsilon maszynowy $1+\varepsilon>1$

Epsilon maszynowy jest wartością określającą precyzję obliczeń numerycznych wykonywanych na liczbach zmiennoprzecinkowych.

Obcięcie

$$\varepsilon = \beta^{1-t}$$

2^(1-3)=1/4=0.25

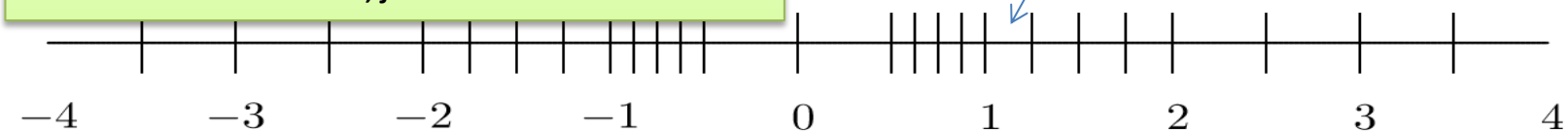
Zaokrąglenie do najbliższej

$$\varepsilon = \frac{1}{2}\beta^{1-t}$$

$$1+\varepsilon>1$$

0.5
0.625
0.75
0.875
1
1.25
1.5
1.75
2
2.5
3
3.5

```
float macheps(void)
{           float e = 1.0f;
            while (1.0f + e / 2.0f > 1.0f)
                        e /= 2.0f;
            return e;}
```



Im mniejsza wartość epsilona maszynowego, tym większa jest względna precyzja obliczeń.

Zero maszynowe, nadmiar i niedomiar

d={0,1}
d0=1
X=(d0+d1/2+d2/4)*2^{-1,0,1}

1 0 0 2^-1	1 0 1 2^-1	1 1 0 2^-1	1 1 1 2^-1
1 0 0 2^0	1 0 1 2^0	1 1 0 2^0	1 1 1 2^0
1 0 0 2^1	1 0 1 2^1	1 1 0 2^1	1 1 1 2^1

3*8=12 wartości liczb dodatnich

krok

0.125

0.25

0.5

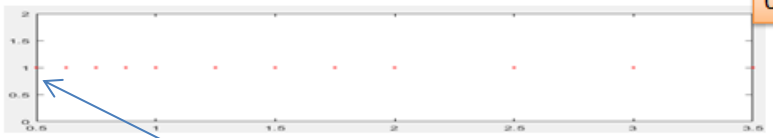
0.5
0.625
0.75
0.875
1
1.25
1.5
1.75
2
2.5
3
3.5

1 0 0 2^-1
1 0 0 2^0
1 0 0 2^1

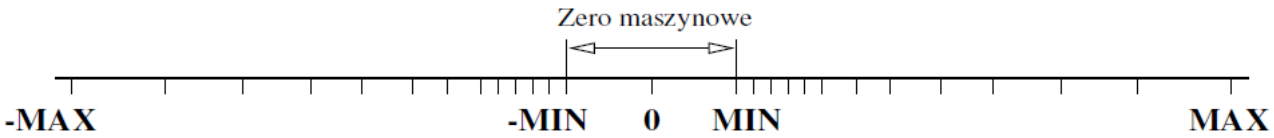
X=1*2^-1=0.5
X=1*2^0=1
X=1*2^1=2

1 0 1 2^-1
1 0 1 2^0
1 0 1 2^1

X=(1+1/4)*2^-1=0.625
X=(1+1/4)*2^0=1.25
X=(1+1/4)*2^1=2.5



Jeśli |x|> MAX, to mówimy o **nadmiarze** (mogą być przerywane obliczenia).
Jeśli |x| < MIN, to rd(x) = 0 i mówimy o **niedomiarze**. Błąd tej reprezentacji jest równy 100%.



Przykład

Jaka jest reprezentacja liczby 9.13 *w arytmetyce fl*, $\beta = 2$, *z mantysą o długości* $t = 6$*? Dla tej arytmetyki* $\epsilon \approx 1.56 \cdot 10^{-2}$

$$x = 9.13 = (1001.0010\ldots)_2 \overset{\text{normalizacja}}{=} (0.10010010\ldots)_2 \cdot 2^4 \xrightarrow{\text{zaokrąglenie}}$$
$$rd(x) = (0.100101)_2 \cdot 2^4. \ |\delta| = |rd(x) - x|/|x| \approx 1.32 \cdot 10^{-2}.$$

$$\boxed{\varepsilon = \frac{1}{2}\beta^{1-t}}$$

>> 1/2*b^(1-t)= 0.015625

Przykład

a = 123456792
a+1 = 123456792
a+2 = 123456792
a+3 = 123456792
a+5 = 123456800
a+8 = 123456800
a+9 = 123456800

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    float a, b, f,c;
    short int x=32767;
    short int y;
    a=123456792;
```

Przykład (cd)

```
a = 92345679212345792
a+1 = 92345679212345792
a+2 = 92345679212345792
a+3 = 92345679212345792
a+5 = 92345679212345792
a+8 = 92345679212345792
a+9 = 92345679212345808
```

```c
#include <stdio.h>
#include <stdlib.h>

        int main()
        {
            double a, b, f,c;
            short int x=32767;
            short int y;
            a=92345679212345800;
```

Własności arytmetyki zmiennoprzecinkowej (fl arytmetyka)

Arytmetyka zmiennoprzecinkowa **nie jest łączna**. To znaczy, że dla x, y i z mogą zachodzić różności:

- $(x + y) + z \neq x + (y + z)$
- $(xy)z \neq x(yz)$,

Nie jest też rozdzielna, czyli może zachodzić różność:

- $x(y + z) \neq (xy) + (xz)$

Innymi słowy, kolejność wykonywania operacji wpływa na końcowy wynik.
Przy obliczeniach zmiennoprzecinkowych występują też:

- zaokrąglenia
- nieprawidłowe operacje
- przepełnienie
- niedomiar

onlinegdb.com/?dest_fid=33258517

OnlineGDB beta
online compiler and debugger for c/c++

Welcome, **Oleksandr Sokolov**

W1OS3

- Create New Project
- My Projects
- Classroom new
- Learn Programming
- Programming Questions
- Upgrade
- Logout

Run | Debug | Stop | Share | Save | { } Beautify | Language C

main.c

```c
#include <stdio.h>
#include <stdlib.h>

        int main()
        {
            float a,b,c,x,y,w;
            a=0.1234567e0;
            b=0.4711325e2;
            c=-b;

            x=(a+b)+c;
            y=a+(b+c);
        //x=a+b;
        //y=0;

```

input

```
Result: a 0.123457
Result: b 47.113251
Result: c -47.113251
Result: x 0.123455
Result: y 0.123457
--------------


...Program finished with exit code 0
Press ENTER to exit console.
```

About • FAQ • Blog • Terms of Use • Contact Us • GDB Tutorial • Credits • Privacy
© 2016 - 2023 GDB Online

W3MNIIOS (1).pptx Niepowodzenie – Błąd pobier... | W3MNIIOS (1).pptx | W3MNIIOS.pptx | FileZilla_3.63.2.1_w....exe | Pokaż wszystkie

Wyszukiwanie | 4°C Duże zachmurzenie | POL 18:11

Arytmetyka zmiennopozycyjna

Dodawanie i odejmowanie

Załóżmy że chcemy dodać lub odjąć dwie dodatnie liczby zmiennoprzecinkowe:

$$x_1 = M_1 B^{E_1} \qquad x_2 = M_2 B^{E_2} \qquad x_1 \pm x_2 = (M_1 \pm M_2 B^{E_2 - E_1}) B^{E_1}$$

Jeśli liczby mają różne wykładniki, to podczas dodawania **mantysa liczby o mniejszym wykładniku musi zostać zdenormalizowana**

Mnożenie i dzielenie

$$3 \cdot 10^2 + 2 \cdot 10^1 =$$

$$x_1 = S_1 M_1 B^{E_1} \text{ i } x_2 = S_2 M_2 B^{E_2}$$

$$3 \cdot 10^2 \times 2 \cdot 10^1 =$$

$$x_1 x_2 = (S_1 S_2)(M_1 M_2) B^{E_1 + E_2}$$

$$x_1 / x_2 = (S_1 S_2)(M_1 / M_2) B^{E_1 - E_2}$$

Błędy operacji elementarnych

- wyniki działań arytmetycznych nie muszą być liczbami maszynowymi – mamy więc *działania zmiennopozycyjne* aproksymujące właściwe działania, np:

$$
\begin{aligned}
x +^* y &\stackrel{\text{def}}{=} rd(x+y) \\
x -^* y &\stackrel{\text{def}}{=} rd(x-y) \\
x \times^* y &\stackrel{\text{def}}{=} rd(x \times y) \\
x /^* y &\stackrel{\text{def}}{=} rd(x/y)
\end{aligned}
$$

Wówczas:

$$
\begin{aligned}
x +^* y &= (x+y)(1+\epsilon_1) \\
x -^* y &= (x-y)(1+\epsilon_2) \\
x \times^* y &= (x \times y)(1+\epsilon_3) \\
x /^* y &= (x/y)(1+\epsilon_4),
\end{aligned}
$$

$$rd(x) = x \cdot (1+\varepsilon)$$

gdzie $|\epsilon_i| \leqslant eps$

Wynik działań zmiennopozycyjnych można traktować jako wynik dokładnych działań wykonanych na zaburzonych danych wejściowych.

Błędy operacji elementarnych (cd)

Reprezentację zmiennopozycyjną liczby x oznaczamy **rd(x)**
Wartość wyrażenia w arytmetyce zmiennopozycyjnej **fl(wyrażenie)**

$fl(a+b)$,
$fl(a-b)$,
$fl(a*b)$,
$fl(a/b)$.

Układ dwójkowy (**d** – ilość bitów reprezentacji liczby)

$$\boldsymbol{fl(a \bullet b)} = \mathrm{rd}(\boldsymbol{a \bullet b})(1+\boldsymbol{\varepsilon}), \text{ gdzie } |\varepsilon| \leq 2^{-d}$$

znak • oznacza jeden z symboli działania: +, -, *, /

$$\left|\frac{\mathrm{rd}(x) - x}{x}\right| \leq 2^{-d}$$

$$\mathrm{rd}(x) = x(1+\varepsilon), \text{ gdzie } |\varepsilon| \leq 2^{-d}$$

Niech r będzie rzeczywistym błędem względnym, tzn.

$$\tilde{x} = x + rx = x(1+r).$$

Przykład

Błąd wynikający z niewystarczająco dużej mantysy można pokazać na przykładzie obliczania sumy dwóch liczb:

$a = 231\ 000\ 000.0$

$b = 0.000\ 000\ 384$

$a+b = 231\ 000\ 000.000\ 000\ 384$

Jeśli długość mantysy wynosi np. 10 cyfr znaczących, to dla dodawania zmiennopozycyjnego otrzymamy:

$fl(a+b) = 231\ 000\ 000.0$

czyli:

$fl(a+b) = a$ (!!!).

$$\boxed{fl(a+fl(b+c)) \neq fl(fl(a+b)+c)}$$

$a+b+c$ Dodawanie z **siedmiocyfrowymi** mantysami

$$a = 0.1234567 \cdot 10^0, b = 0.4711325 \cdot 10^4, c = -b$$

Sposób A

$\boxed{fl(a+fl(b+c))}$

$$fl(b+c) = 0,$$

$$fl(a+fl(b+c)) = a = 0.1234567 \cdot 10^0$$

Sposób B

$\boxed{fl(fl(a+b)+c)}$

a	0.0000123	$4567 \cdot 10^4$
$+\ b$	0.4711325	$\cdot 10^4$
$fl(a+b)$	0.4711448	$\cdot 10^4$
$+c$	-0.4711325	$\cdot 10^4$
$fl(fl(a+b)+c)$	0.0000123	$\cdot 10^4$

$$fl(a+fl(b+c)) = 0.1234567 \cdot 10^0.$$

$$fl(fl(a+b)+c) = 0.0000123 \cdot 10^4 = 0.1230000 \cdot 10^0$$

Przykład

$$w=a^2-b^2$$

mantysa jest reprezentowana na **d bitach**

błędy reprezentacji wynoszą $\varepsilon_i \leq 2^{-d}$

Sposób 1

$x=a*a,\ y=b*b,\ w=x-y$

$$fl(x) = (a*a)(1+\varepsilon_1),$$
$$fl(y) = (b*b)(1+\varepsilon_2),$$
$$fl(w) = [(a*a)(1+\varepsilon_1) - (b*b)(1+\varepsilon_2)](1+\varepsilon_3) =$$
$$= (a^2 - b^2)\left[\frac{a^2\varepsilon_1 - b^2\varepsilon_2}{a^2 - b^2} + 1\right](1+\varepsilon_3) = (a^2 - b^2)(1+\eta_1),$$

gdzie:

$$(1+\eta_1) = \left[1 + \frac{a^2\varepsilon_1 - b^2\varepsilon_2}{a^2 - b^2}\right](1+\varepsilon_3) =$$
$$= \left[1 + \frac{a^2\varepsilon_1 - b^2\varepsilon_2}{a^2 - b^2}\right] + \varepsilon_3 + O(\varepsilon_1, \varepsilon_1, \varepsilon_1),$$

a $O(\varepsilon_1, \varepsilon_1, \varepsilon_1)$ jest pomijalnie małe.

Jeśli $a^2 \approx b^2$ (mianownik bliski 0) oraz ε_1 i ε_2 mają przeciwne znaki to błąd η_1 może być bardzo duży.

Sposób 2

$x=a+b,\ y=a-b,\ w=x*y$

$$fl(x) = (a+b)(1+\varepsilon_1),$$
$$fl(y) = (a-b)(1+\varepsilon_2),$$
$$fl(w) = [(a+b)(1+\varepsilon_1)(a-b)(1+\varepsilon_2)](1+\varepsilon_3) = (a^2-b^2)(1+\eta_2),$$

gdzie:

$$(1+\eta_2) = (1+\varepsilon_1)(1+\varepsilon_2)(1+\varepsilon_3) =$$
$$= 1+\varepsilon_1+\varepsilon_2+\varepsilon_3+\varepsilon_1\varepsilon_2+\varepsilon_1\varepsilon_3+\varepsilon_3\varepsilon_2+\varepsilon_1\varepsilon_2\varepsilon_3 \approx 1+\varepsilon_1+\varepsilon_2+\varepsilon_3.$$

W tym przypadku, niezależnie od wartości a i b, mamy zawsze:

$$\eta_2 \le 3\cdot 2^{-d}.$$

Błąd dla drugiego algorytmu jest mniejszy i nie zależy od wartości a i b.

Przykład

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    float a,b,x,y,w;
    a=123456;
    b=123455;
    x=a*a;
    y=b*b;
    w=x-y;

        printf("Result: a %f\n", a);
        printf("Result: b %f\n", b);
        printf("Result: x %f\n", x);
        printf("Result: y %f\n", y);
        printf("Result: w %f\n", w);
        printf("--------------\n");

    x=a+b;
    y=a-b;
    w=x*y;

        printf("Result: a %f\n", a);
        printf("Result: b %f\n", b);
        printf("Result: x %f\n", x);
        printf("Result: y %f\n", y);
        printf("Result: w %f\n", w);

        return 0;
}
```

```
Result: a 123456.000000
Result: b 123455.000000
Result: x 15241383936.000000
Result: y 15241137152.000000
Result: w 246784.000000
--------------
Result: a 123456.000000
Result: b 123455.000000
Result: x 246911.000000
Result: y 1.000000
Result: w 246911.000000
```

```
Result: a 123456.000000
Result: b 1.000000
Result: x 15241383936.000000
Result: y 1.000000
Result: w 15241383936.000000
--------------
Result: a 123456.000000
Result: b 1.000000
Result: x 123457.000000
Result: y 123455.000000
Result: w 15241383936.000000
```

MNII\programOS\bledy\w1OS1.c

Algorytm numerycznie stabilny i poprawny

Niestabilność numeryczna powstaje wówczas, kiedy **mały błąd numeryczny w trakcie dalszych obliczeń powiększa się** (np. przemnaża się) i powoduje duży błąd wyniku.

Obliczyć dla $n = 0,1,..,15$ całki: $y_n = \int_0^1 \frac{x^n}{x+5}\,\mathrm{d}x$.

Zauważmy, że:

$$y_n + 5y_{n-1} = \int_0^1 \frac{x^n + 5x^{n-1}}{x+5}\,\mathrm{d}x = \int_0^1 \frac{x^{n-1}(x+5)}{x+5}\,\mathrm{d}x = \left.\frac{x^n}{n}\right|_0^1 = \frac{1}{n}.$$

Otrzymujemy wobec tego wzór rekurencyjny:

$$y_n + 5y_{n-1} = \frac{1}{n},$$

Algorytm 1

Korzystając z wzoru:

$$y_n = \frac{1}{n} - 5y_{n-1},$$

mamy:

$$y_0 = \int_0^1 \frac{\mathrm{d}x}{x+5} = \ln(x+5)\ \Big|_0^1 = \ln 6 - \ln 5 \approx 0.18232156.$$

Przyjmując $y_0 = 0.182321556$, obliczamy kolejno:

$y_1 = 1 - 5y_0 \approx 0.0884$,

$y_2 = 1/2 - 5y_1 \approx 0.0580$,

$y_3 = 1/3 - 5y_2 \approx 0.0431$,

.....

$y_{10} = 1/9 - 5y_9 \approx 0.0040$,

$y_{11} \approx 1/10 - 5y_{10} \approx -0.3071$ (wynik błędny – wartość całki oznaczonej ujemna?).

Algorytm 1(cd)

Powodem otrzymania złego wyniku jest to, że błąd zaokrąglenia ε wartości y_0 jest mnożony przez -5 przy obliczaniu y_1. Tak więc wartość y_1 jest obarczona błędem -5ε. Ten błąd tworzy błąd 25ε w y_2, -125ε w y_3 itd. Nakładają się na to błędy zaokrąglenia popełniane w kolejnych krokach obliczeń, mające jednak stosunkowo małe znaczenie. Podstawiając $y_0 = \ln 6 - \ln 5$ popełniamy mniejszy błąd zaokrąglenia, który także powoduje duże zniekształcenie wyniku obliczeń y_i, dla $i > 16$. Oczywiście otrzymywane wyniki zależą także od precyzji, z jaką przeprowadzano obliczenia. Dla podwójnej precyzji obliczeń (reprezentacja liczby na 8 bajtach), wyraźnie błędne wyniki występują dla $i > 20$.

Algorytm 2

Ten sam ciąg całek możemy wyznaczyć inaczej. Jeśli przekształcimy wzór na zależność rekurencyjną tak, żeby obliczać w kolejnych iteracjach elementy ciągu w drugą stronę, mamy:

$$y_{n-1} = \frac{1}{5n} - \frac{1}{5} y_n .$$

Dzięki temu w każdym kroku błąd będzie dzielony przez -5. Ponieważ y_n maleje gdy n rośnie, możemy przypuszczać, że dla dużych n, y_n maleje wolno. Wobec tego przyjmując $y_{16} \approx y_{17}$ i korzystając z wzoru:

$$y_{16} = \frac{1}{5 \cdot 17} - \frac{1}{5} y_{17} ,$$

otrzymujemy:

$$y_{16} \approx \frac{1}{5 \cdot 17} - \frac{1}{5} y_{16} \Rightarrow y_{16} \approx \frac{1}{5 \cdot 17} \cdot \frac{5}{6} \approx 0.009803921 .$$

Algorytm 2 (cd)

Następnie obliczamy:

$$y_{15} = \frac{1}{5 \cdot 16} - \frac{1}{5} y_{16} \approx 0.0113,$$

$$y_{14} \approx 0.0120,$$

$$y_{13} \approx 0.0129,$$

.....

$$y_0 \approx 0.18232156 \text{ (wynik poprawny).}$$

$$y_0 = \int_0^1 \frac{dx}{x+5} = \ln(x+5) \Big|_0^1 = \ln 6 - \ln 5 \approx 0.18232156.$$

W przypadku algorytmu 1 mamy do czynienia z niestabilnością numeryczną, bowiem małe błędy popełniane na pewnym etapie obliczeń rosną w następnych etapach i istotnie zniekształcają ostateczne wyniki (nawet co do znaku!).

Stabilność a poprawność

Maksymalny przewidywalny błąd wynikły wyłącznie z przeniesienia błędu reprezentacji danych na wynik obliczeń nazywamy **optymalnym poziomem błędu** danego zadania w arytmetyce t-cyfrowej.

Algorytm stabilny gwarantuje otrzymanie wyniku akceptowanego z poziomem błędu **tego samego rzędu, co optymalny poziom błędu**. Rozwiązanie obliczone algorytmem numerycznie poprawnym jest nieco zaburzonym rozwiązaniem zadania o nieco zaburzonych danych, tzn. **jeśli dane są obarczone błędem, to i wynik jest obarczony porównywalnym błędem**.

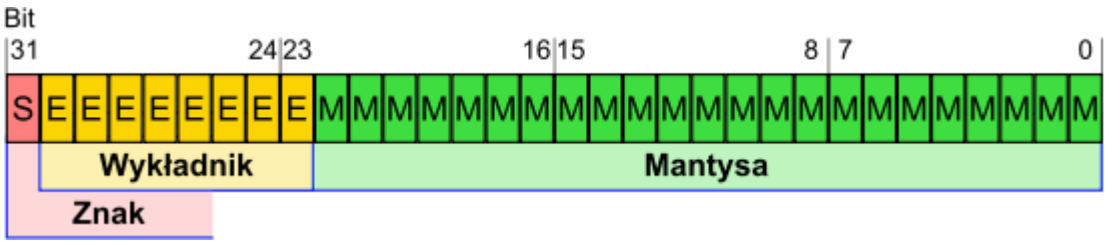
Stabilność jest minimalną własnością, jakiej wymagamy od algorytmu, **poprawność** - maksymalną własnością jakiej możemy oczekiwać od zastosowanego algorytmu

Standard IEEE-754

Reprezentacja zmiennoprzecinkowa IEEE-754 single

$$g = (-1)^s * 1.f \times 2^{e-127}$$

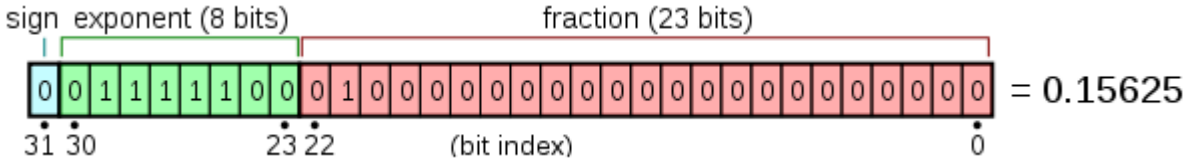
-127!



Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias[6]	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1=15$	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1=127$	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1=1023$	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1=16383$	−16382	+16383	
decimal32		10	7	7	7.58	96	101	−95	+96	not basic
decimal64		10	16	16	9.58	384	398	−383	+384	
decimal128		10	34	34	13.58	6144	6176	−6143	+6144	

https://pl.wikipedia.org/wiki/IEEE_754

Przykład



$$(-1)^{b_{31}} \times (1.b_{22}b_{21}\ldots b_0)_2 \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127}$$

$$\boxed{\text{value} = (-1)^{\text{sign}} \times \left(1 + \sum_{i=1}^{23} b_{23-i}2^{-i}\right) \times 2^{(e-127)}}$$

- $\text{sign} = b_{31} = 0$,
- $(-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\}$,
- $e = b_{30}b_{29}\ldots b_{23} = \sum_{i=0}^{7} b_{23+i}2^{+i} = 124 \in \{1, \ldots, (2^8 - 1) - 1\} = \{1, \ldots, 254\}$,
- $2^{(e-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \ldots, 2^{127}\}$,
- $1.b_{22}b_{21}\ldots b_0 = 1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1 + 2^{-23}, \ldots, 2 - 2^{-23}\} \subset [1; 2 - 2^{-23}] \subset [1; 2)$.

$$\text{value} = (+1) \times 1.25 \times 2^{-3} = +0.15625.$$

Przykład

Przekształcamy liczbę dziesiętną do postaci dwójkowej:

$$5,375 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} \rightarrow 101.011.$$

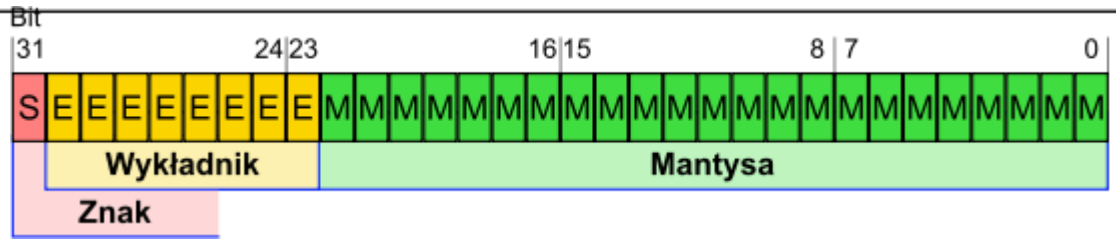
Otrzymaną liczbę normalizujemy:

$$101.011 \times 2^0 = 1.01011 \times 2^2.$$

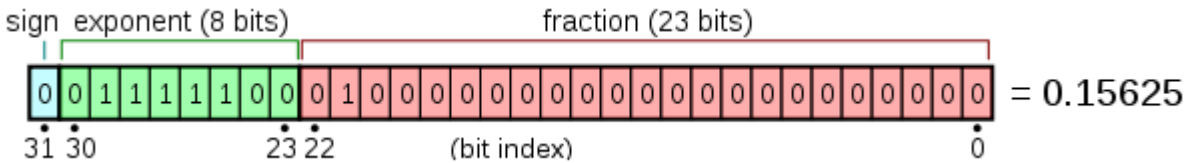
pomijamy wiodącą jedynkę w mantysie 01011

obliczamy wykładnik 2 + 127 = 129 → 10000001

znak	wykładnik	mantysa
0	10000001	01011000000000000000000



Zapis 10-tnej liczby



12.375

$0.375 \times 2 = 0.750 = 0 + 0.750 \Rightarrow b_{-1} = 0,$

$0.750 \times 2 = 1.500 = 1 + 0.500 \Rightarrow b_{-2} = 1$

$0.500 \times 2 = 1.000 = 1 + 0.000 \Rightarrow b_{-3} = 1,$

$(0.375)_{10} \longrightarrow (0.011)_2$

$(12.375)_{10} = (12)_{10} + (0.375)_{10} = (1100)_2 + (0.011)_2 = (1100.011)_2$

$$(1.x_1 x_2 \ldots x_{23})_2 \times 2^e$$

$$(12.375)_{10} = (1.100011)_2 \times 2^3$$

127 = 0111 1111

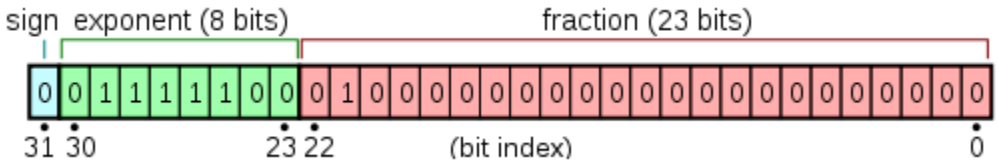
130 = 1000 0010

https://www.rapidtables.com/convert/number/decimal-to-binary.html?x=12.375

Przykłady

- $1 + 2^{-23} \approx 1.000\,000\,119$,
- $2 - 2^{-23} \approx 1.999\,999\,881$,
- $2^{-126} \approx 1.175\,494\,35 \times 10^{-38}$,
- $2^{+127} \approx 1.701\,411\,83 \times 10^{+38}$.

Zadania

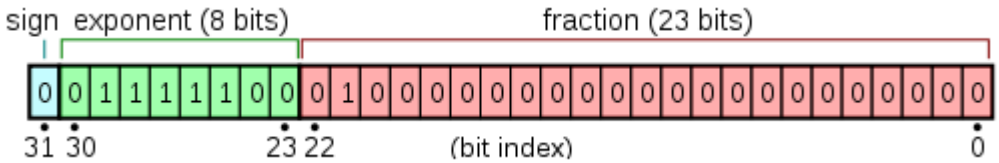


1_{10}

0.25_{10}

0.375_{10}

Odpowiedzi



$$(1)_{10} = (1.0)_2 \times 2^0$$

127 = 0111 1111

0 = 000...0

0-01111111-00000000000000000000000 = 3f800000$_H$

$$(0.25)_{10} = (1.0)_2 \times 2^{-2}$$

127+(−2)= 125 = 0111 1101

0 = 000...0

0-01111101-00000000000000000000000 = 3e800000H

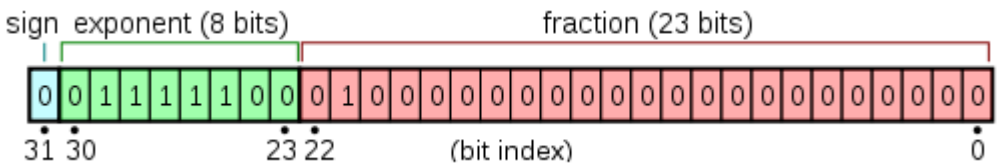
$$0.375 = (1.1)_2 \times 2^{-2}$$

127+(−2)= 125 = 0111 1101

1 = 10000..0

0-01111101-10000000000000000000000 = 3ec00000H

Z 2-go do 10-tnej



$41c8\ 0000_{16} = 0100\ 0001\ 1100\ 1000\ 0000\ 0000\ 0000\ 0000_2$

Sign bit: 0

Exponent: $1000\ 0011_2 = 83_{16} = 131$

$131 - 127 = 4$

Significand: $100\ 1000\ 0000\ 0000\ 0000\ 0000_2 = 480000_{16}$

bit 23 = 0.5
bit 22 = 0.25
bit 21 = 0.125
bit 20 = 0.0625
bit 19 = 0.03125
..
bit 0 = 0.00000011920928955078125

$$1 + \overset{23}{0.5} + \overset{20}{0.0625} = 1.5625$$

$1.5625 \times 2^4 = 25$

41c8 0000 = 25

Przykłady

3f80 0000 = 0 01111111 00000000000000000000000 = 1
c000 0000 = 1 10000000 00000000000000000000000 = −2
3eaa aaab = 0 01111101 01010101010101010101011 ≈ 1/3

7f7f ffff = 0 11111110 11111111111111111111111 = $(1 - 2^{-24}) \times 2^{128} \approx 3.402823466 \times 10^{38}$ (max finite positive value in single precision)

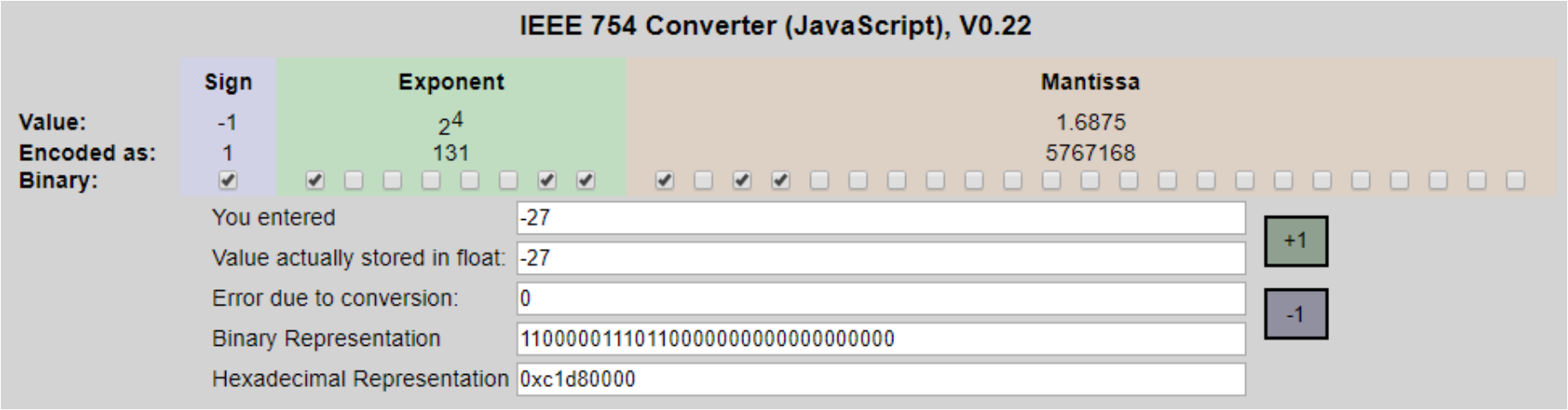
0080 0000 = 0 00000001 00000000000000000000000 = $2^{-126} \approx 1.175494351 \times 10^{-38}$ (min normalized positive value in single precision)

0000 0000 = 0 00000000 00000000000000000000000 = 0
8000 0000 = 1 00000000 00000000000000000000000 = −0

7f80 0000 = 0 11111111 00000000000000000000000 = infinity
ff80 0000 = 1 11111111 00000000000000000000000 = −infinity

Online Konwerter

https://www.h-schmidt.net/FloatConverter/IEEE754.html



Cechy standardu

Precision limits on integer values [edit]

- Integers in $[-16777216, 16777216]$ can be exactly represented
- Integers in $[-33554432, -16777217]$ or in $[16777217, 33554432]$ round to a multiple of 2
- Integers in $[-2^{26}, -2^{25} - 1]$ or in $[2^{25} + 1, 2^{26}]$ round to a multiple of 4
- ....
- Integers in $[-2^{127}, -2^{126} - 1]$ or in $[2^{126} + 1, 2^{127}]$ round to a multiple of 2^{103}
- Integers in $[-2^{128} + 2^{104}, -2^{127} - 1]$ or in $[2^{127} + 1, 2^{128} - 2^{104}]$ round to a multiple of 2^{127-23}
- Integers larger or equal than 2^{128} or smaller or equal than -2^{128} are rounded to "infinity".

a=123456792

a = 33554400
a+1 = 33554400
a+2 = 33554402

```c
#include <stdio.h>
#include <stdlib.h>

        int main()
        {
            float a, b, f,c;
            short int x=32767;
            short int y;
            a=123456792;
            a=33554400;
```

Standard IEEE-754. Reprezentacja bitowa

precyzja	znak	wykładnik	część ułamkowa	przesunięcie
pojedyncza	1 [31]	8 [30-23]	23 [22-00]	127
podwójna	1 [63]	11 [62-52]	52 [51-00]	1023

- reprezentacja binarna – podstawa równa 2
- znak – 0 - liczba dodatnia, 1 - liczba ujemna
- wykładnik i przesunięcie – rzeczywista cecha = wykładnik - przesunięcie
- mantysa
  - *postać znormalizowana liczby* – w mantysie przecinek po pierwszej niezerowej cyfrze ($1 \leqslant m < 10$)
    Przykład: $1.25 * 10^2$, nie $0.125 * 10^3$ ani $125 * 10^0$
  - w systemie dwójkowym niezerowa znaczy jedynka
  - pierwszy bit mantysy zawsze równy 1 – ukryty, reszta to część ułamkowa

Przykład $s \times M \times B^{e-E}$

	sign bit	8-bit exponent	this bit could be "phantom" / 23-bit mantissa	
½ =	0	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(a)
3 =	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(b)
¼ =	0	0 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(c)
10^{-7} =	0	0 1 1 0 1 0 0 1	1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 0	(d)
=	0	1 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(e)
$3 + 10^{-7}$ =	0	1 0 0 0 0 0 1 0	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	(f)

Figure 1.3.1. Floating point representations of numbers in a typical 32-bit (4-byte) format. (a) The number $1/2$ (note the bias in the exponent); (b) the number 3; (c) the number $1/4$; (d) the number 10^{-7}, represented to machine accuracy; (e) the same number 10^{-7}, but shifted so as to have the same exponent as the number 3; with this shifting, all significance is lost and 10^{-7} becomes zero; shifting to a common exponent must occur before two numbers can be added; (f) sum of the numbers $3 + 10^{-7}$, which equals 3 to machine accuracy. Even though 10^{-7} can be represented accurately by itself, it cannot accurately be added to a much larger number.

Przykłady

23.75

- postać binarna: $10111.11 = 2^4 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2}$
- po normalizacji: $1.011111 * 2^4 = 1.011111 * 2^{100}$
- znak 0
  wykładnik z przesunięciem $4 + 127 = 131 = 10000011$
  część ułamkowa 011111
- ostatecznie mamy 0 10000011 01111100000000000000000
  czyli 01000001 10111110 00000000 00000000
  szesnastkowo 0x41BE0000

-0.7

- postać binarna: $-0.1011001100110011...$
- znak = 0, wykładnik $= -1$, $127 - 1 = 126 = 01111110$
- ostatecznie mamy 1 01111110 01100110011001100110011
  czyli 10111111 00110011 00110011 00110011
  szesnastkowo 0xBF333333

prefiks 0x wskazuje, że liczba jest napisana w kodzie szesnastkowym

Reprezentacja liczb specjalnych

0	00000000	00000000000000000000000	0
1	00000000	00000000000000000000000	-0
0	11111111	00000000000000000000000	$+\infty$
1	11111111	00000000000000000000000	$-\infty$
0	11111111	00000100000000000000000	NaN
1	11111111	00100010001001010101010	NaN

Not a Number

$$\infty + (-\infty)$$
$$0 \times \infty$$
$$0/0, \infty/\infty$$
$$x \ mod \ 0, \ \infty \ mod \ y$$
$$\sqrt{x}, \ x < 0$$

Standard IEEE-754. Wartości specjalne

znak	wykładnik (w)	cz. ułamkowa (u)	wartość
0	00..00	00..00	+0
0	00..00	00..01 – 11..11	dodatnia zdenorm. $0.u * 2^{-p+1}$
0	00..01 – 11..10	00..00 – 11..11	dodatnia znorm. $1.u * 2^{w-p}$
0	11..11	00..00	+Infinity (nieskończoność)
0	11..11	00..01 – 01..11	SNaN (Signalling Not A Number)
0	11..11	10..00 – 11..11	QNaN (Quiet Not A Number)
1	00..00	00..00	-0
1	00..00	00..01 – 11..11	ujemna zdenorm. $-0.u * 2^{-p+1}$
1	00..01 – 11..10	00..00 – 11..11	ujemna znorm. $-1.u * 2^{w-p}$
1	11..11	00..00	-Infinity (nieskończoność)
1	11..11	00..01 – 01..11	SNaN
1	11..11	10..00 – 11..11	QNaN

Standard IEEE-754. Operacje specjalne

Operacja	Wynik
n / ±Infinity	0
±Infinity x ±Infinity	±Infinity
±nonzero / 0	±Infinity
Infinity + Infinity	Infinity

Operacja	Wynik
±0 / ±0	NaN
Infinity - Infinity	NaN
±Infinity / ±Infinity	NaN
±Infinity x 0	NaN

Implementacje w C++

Typ	Bity	Największa liczba
`float`	32	3.40282×10^{38}
`double`	64	1.79769×10^{308}
`long double`	96	1.18974×10^{4932}

Niedokładności

Wiele dziwnych na pierwszy rzut oka wyników bierze się stąd, że obliczenia zmiennopozycyjne wykonywane są w bazie B = 2, natomiast na zewnątrz liczby reprezentowane są najczęściej w bazie B = 10. Dlatego pisząc w kodzie programu np. x=0.01 intuicyjnie oczekujemy, że liczba 0,01 będzie reprezentowana dokładnie, a tak niestety nie jest. Aby się o tym przekonać, rozważmy następujący program:

```
#include <iostream>

int main()
{
    float x=0.01;
    if(100.0*x==1.0)
      std::cout << "Równe :)\n";
    else
      std::cout << "Nierówne :(\n";
}
```

```
Nierówne :(
```

Ponieważ liczba 0, 01 nie ma dokładniej reprezentacji w standardzie IEEE, przybliżana jest najbliższą liczbą maszynową (czyli taką, która daje się przedstawić dokładnie). Dlatego równość nie jest spełniona.

Niedokładności (c.d.)

```
#include <iostream>

int main()
{
   float x=77777.0, y=7.0;
   float y1 = 1.0/y;
   float z= x/y;
   float z1 = x*y1;
   if(z==z1)
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}
```

Po wykonaniu tego programu znowu okaże się, wbrew oczekiwaniom, że wartości zmiennych z i z1 nie są równe. Podobnie, jak w poprzednim przykładzie, przyczyną jest to, że y = 1.0/7.0 nie ma dokładniej reprezentacji w standardzie IEEE.

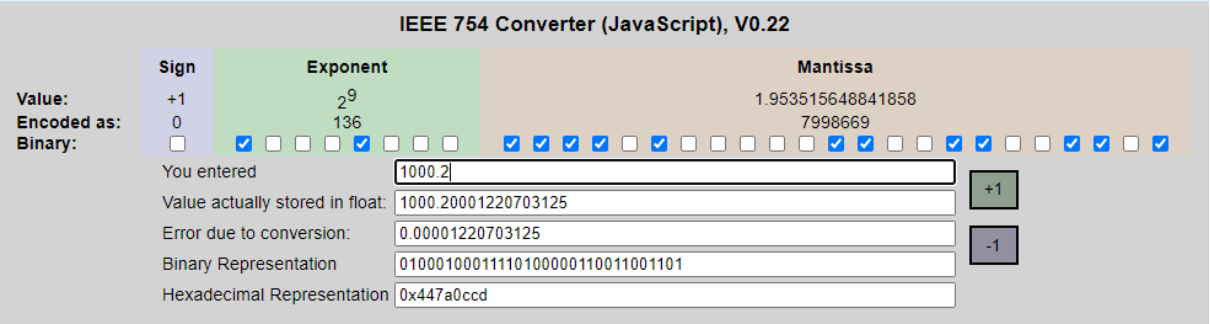
Niedokładności (c.d.)

```
#include <iostream>

int main()
{
    float y=1000.2;
    float x=y-1000.0;
    std::cout << x <<"\n";
}
```

Liczba 1000, 0 ma dokładną reprezentację zmiennopozycyjną, natomiast 1000, 2 daje się przedstawić tylko w przybliżeniu. Dlatego komputerowy wynik powyższego działania będzie się różnił trochę od wartości dokładnej 0, 2:

```
0.200012
```



Rozszerzona precyzja

Rodzina procesorów Intela należy do systemów o rozszerzonej precyzji (ang. extended-based systems) , które wewnętrznie zapisują liczby z dokładnością większą, niż wymaga tego używany typ danych. Pozwala to na ogół liczyć dokładniej niż na architekturach pozbawionych tej właściwości (np. większość procesorów RISC), czasami prowadzi jednak do pewnych niepożądanych efektów ubocznych:

```
#include <iostream>

int main()
{
   float a=10.0,b=3.0,x=10.0,y=3.0;
   float c=a/b;
   float z=c-(x/y);
   std::cout << z <<"\n";
}
```

Iloraz a/b liczony jest w rozszerzonej precyzji. Wynik zostaje przekształcony do pojedynczej precyzji i zapamiętany w zmiennej c. Jeżeli przy tym przekształceniu następuje utrata dokładności, z będzie raczej różne od zera:

```
-7.94729e-08
```

Porównywanie liczb zmiennopozycyjnych

```
OS2.c
# include  "stdio.h"
int main ()
{
float x =3.0/7.0;
if ( x ==3.0/7.0)
printf( " Równe :)\ n " );
else
printf( " Nierówne :(\ n ") ;
}
```

Powodem takiego zachowania programu jest znowu rozszerzona precyzja, z jaką liczby przechowywane są w wewnętrznych rejestrach i jej utrata przy konwersji na typ używany w programie. Dlatego należy unikać bezpośrednich porównań liczb zmiennoprzecinkowych, a jeżeli algorytm wymaga od nas takiego porównywania, bezpieczniej jest traktować liczby jako równe, jeżeli moduł ich różnicy jest mniejszy od małej, z góry zadanej wartości:

```
#include <iostream>
#include <cmath>

int main()
{
   float eps=1.0e-6;
   float x=3.0/7.0;
   if(fabs(x-3.0/7.0)<eps)
     std::cout << "Równe :)\n";
   else
     std::cout << "Nierówne :(\n";
}

Równe :)
```

Denormalizacja

```
>> x =10^10; y = x ; z =1/ x ;
>> x+z-y

ans =     0

>> x-y+z

ans =   1.0000e-10
```

Chociaż oba wyrażenia są matematycznie równoważne, dają numerycznie różne wyniki. Przyczyną jest znowu skończona ilość bitów w mantysie. Jeżeli dodajemy do siebie dwie liczby o różnych wykładnikach, muszą one zostać sprowadzone do wspólnej cechy. Na ogół odbywa się to przez **denormalizację mniejszej z liczb**.

Jeżeli rzędy tych liczb różnią się od siebie o więcej niż 52 bity, wówczas w procesie denormalizacji mniejsza liczba zostanie zastąpiona zerem.

Tak jest w pierwszym z powyższych wyrażeń.

Przykład. Utrata cyfr znaczących

$$f_1(x) = \sqrt{x}\left(\sqrt{x+1} - \sqrt{x}\right) = f_2(x) = \frac{\sqrt{x}}{\sqrt{x+1} + \sqrt{x}}.$$

```
x=                1, f1(x)=0.414213562373095145, f2(x)=0.414213562373095090
x=               10, f1(x)=0.488088481701514754, f2(x)=0.488088481701515475
x=              100, f1(x)=0.498756211208899458, f2(x)=0.498756211208902733
x=             1000, f1(x)=0.499875062461021868, f2(x)=0.499875062460964859
x=            10000, f1(x)=0.499987500624854420, f2(x)=0.499987500624960890
x=           100000, f1(x)=0.499998750005928860, f2(x)=0.499998750006249937
x=          1000000, f1(x)=0.499999875046341913, f2(x)=0.499999875000062488
x=         10000000, f1(x)=0.499999987401150925, f2(x)=0.499999987500000576
x=        100000000, f1(x)=0.500000005558831617, f2(x)=0.499999998749999952
x=       1000000000, f1(x)=0.500000077997506343, f2(x)=0.499999999874999990
x=      10000000000, f1(x)=0.499999441672116518, f2(x)=0.499999999987500054
x=     100000000000, f1(x)=0.500004449631168080, f2(x)=0.499999999998750000
x=    1000000000000, f1(x)=0.500003807246685028, f2(x)=0.499999999999874989
x=   10000000000000, f1(x)=0.499194546973835973, f2(x)=0.499999999999987510
x=  100000000000000, f1(x)=0.502914190292358398, f2(x)=0.499999999999998723
```

$$\begin{aligned}
\sqrt{x+1} + \sqrt{x} &= 63245553.20336761 \\
\sqrt{x+1} - \sqrt{x} &\simeq 0.00000002
\end{aligned}$$

Przykład

```
#include "stdio.h"
int
main(int argc, char *argv[])
{
        float a, b, f;
        a=123456789;
        b=123456788;
        f=a-b;
printf("Result: %f\n", f);
        return 0;

}
```

```
Result: 8.000000
```

Decimal Representation	1.23456792E8
Binary Representation	01001100111010110111100110100011
Hexadecimal Representation	0x4ceb79a3
After casting to double precision	1.23456792E8

123456789 =4CEB79A3hex(ieee)=123456792(dec)

123456788 =4CEB79A2hex(ieee)=123456784(dec)

http://www.h-schmidt.net/FloatConverter/IEEE754.html

The first 2398 decimal digits of π

3.14159265358979323846264338327950288419716939937510582097494459230781640 62
86208998628034825342117067982148086513282306647093844609550582231725359408 1
28481117450284102701938521105559644622948954930381964428810975665933446128 4
75648233786783165271201909145648566923460348610454326648213393607260249141 2
73724587006606315588174881520920962829254091715364367892590360011330530548 8
20466521384146951941511609433057270365759591953092186117381932611793105118 5
48074462379962749567351885752724891227938183011949129833673362440656643086 0
21394946395224737190702179860943702770539217176293176752384674818467669405 1
32000568127145263560827785771342757789609173637178721468440901224953430146 5
49585371050792279689258923542019956112129021960864034418159813629774771309 9
60518707211349999998372978049951059731732816096318595024459455346908302642 5
22308253344685035261931188171010003137838752886587533208381420617177669147 3
03598253490428755468731159562863882353787593751957781857780532171226806613 0
01927876611195909216420198938095257201065485863278865936153381827968230301 9
52035301852968995773622599413891249721775283479131515574857242454150695950 8
29533116861727855889075098381754637464939319255060400927701671139009848824 0
12858361603563707660104710181942955596198946767837449448255379774726847104 0
47534646208046684259069491293313677028989152104752162056966024058038150193 5
11253382430035587640247496473263914199272604269922796782354781636009341721 6
41219924586315030286182974555706749838505494588586926995690927210797509302 9
55321165344987202755960236480665499119881834797753566369807426542527862551 8
18417574672890977772793800081647060016145249192173217214772350141441973568 5
48161361157352552133475741849468438523323907394143334547762416862518983569 4
85562099219222184272550254256887671790494601653466804988627232791786085784 3
83827967976681454100953883786360950680064225125205117392984896084128488626 9
45604241965285022210661186306744278622039194945047123713786960956364371917 2
87467764657573962413890865832645995813390478027590099465764078951269468398 3
52595709825822620522489407726719478268482601476990902640136394437455305068 2
03496252451749399651431429809190659250937221696461515709858387410597885959 7
72975498930161753928468138268683868942774155991855925245953959431049972524 6
80845987273644695848653836736222626099124608051243884390451244136549762780 7
97715691435997700129616089441694868555848406353422072225828488648158456028 5

Zadania

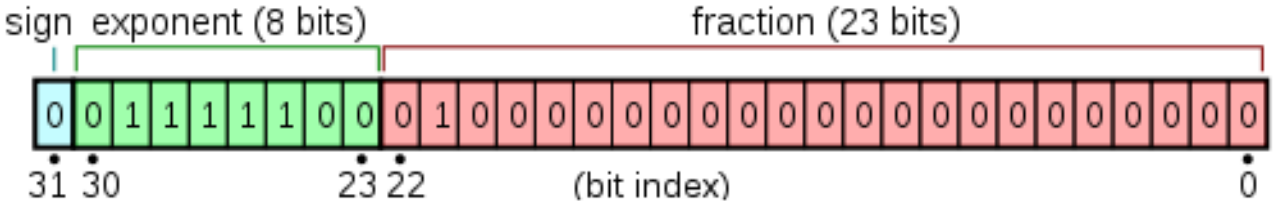
Obliczyć wartość dziesiętną liczby w systemie U2:

$$1001_{U2}$$

Obliczyć wartość dziesiętną liczb zmiennoprzecinkowych :

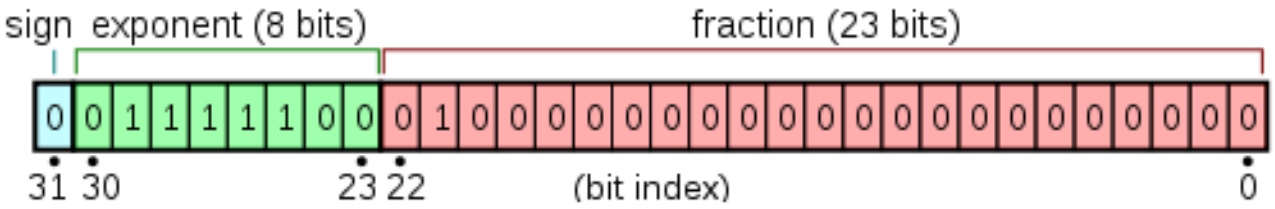
$$11000001110110000000000000000000_{(IEEE\ 754)}$$

$$01000001010100000000000000000000_{(IEEE\ 754)}$$



Odpowiedź

1100000111011000000000000000000(IEEE 754)



1 **10000011** **10110000000000000000000**

z cecha bity ułamkowe mantysy

z = 1 - liczba jest ujemna

c = 10000011(BIAS=127) = 131 **- 127** = 4

m = 01,10110000000000000000000(U1) = 1+11/16=27/16

L(IEEE 754) = (-1)z*m*2^c =

(-1)1 × (1+11/16) × 2^4 = - 27/16 × 2^4= -27(10)

1100000111011000000000000000000(IEEE 754) = -27(10).

13

$01000001010100000000000000000000_{\text{(IEEE 754)}}$